Faster Variational Execution with Transparent Bytecode Transformation

CHU-PAN WONG, Carnegie Mellon University, USA
JENS MEINICKE, Carnegie Mellon University, USA and University of Magdeburg, Germany
LUKAS LAZAREK, Northwestern University, USA
CHRISTIAN KÄSTNER, Carnegie Mellon University, USA

Variational execution is a novel dynamic analysis technique for exploring highly configurable systems and accurately tracking information flow. It is able to efficiently analyze many configurations by aggressively sharing redundancies of program executions. The idea of variational execution has been demonstrated to be effective in exploring variations in the program, especially when the configuration space grows out of control. Existing implementations of variational execution often require heavy lifting of the runtime interpreter, which is painstaking and error-prone. Furthermore, the performance of this approach is suboptimal. For example, the state-of-the-art variational execution interpreter for Java, VarexJ, slows down executions by 100 to 800 times over a single execution for small to medium size Java programs. Instead of modifying existing JVMs, we propose to transform existing bytecode to make it variational, so it can be executed on an unmodified commodity JVM. Our evaluation shows a dramatic improvement on performance over the state-of-the-art, with a speedup of 2 to 46 times, and high efficiency in sharing computations.

CCS Concepts: • **Software and its engineering → Feature interaction**; **Dynamic analysis**; **Software testing and debugging**;

Additional Key Words and Phrases: Java Virtual Machine, Bytecode Transformation, Variational Execution, Configurable System

ACM Reference Format:
Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. 2018. Faster Variational Execution with Transparent Bytecode Transformation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 117 (November 2018), 30 pages. https://doi.org/10.1145/3276487

1 INTRODUCTION

Computer programs often come with variations that allow programs to act differently according to a user's need. A classic example of variation is command-line options or configuration files, which often trigger new behaviors or tweak existing functionalities. Configuration options are widely used because they provide flexible extension points for adding new features. However, this flexibility often comes at the cost of potential feature conflicts. Feature conflicts arise when one feature interferes with another in an unintended way (also known as feature interaction problem [Calder et al. 2003; Nhlabatsi et al. 2008]). Problems similar to feature interactions have also been studied in other contexts, such as security policies, where the sensitivity of a program to various private values is explored by comparing different executions varying in privacy levels for values [Austin

Authors' addresses: Chu-Pan Wong, Carnegie Mellon University, USA; Jens Meinicke, Carnegie Mellon University, USA, University of Magdeburg, Germany; Lukas Lazarek, Northwestern University, USA; Christian Kästner, Carnegie Mellon University, USA.

2475-1421/2018/11-ART117
https://doi.org/10.1145/3276487

and Flanagan 2012; Austin et al. 2013]. To tackle these problems, we need a principled yet efficient way of detecting and managing *interactions of variations*.

There are several challenges posed by *interactions of variations*. First, the number of possible interactions is exponential to the number of variations, making exhaustive testing of all configurations unrealistic in most practical cases. Second, interactions are often impossible to foresee, for example in the context of plugin-based systems where plugins are often developed independently by different developers, making it easy to miss interactions when sampling only few configurations [Medeiros et al. 2015; Nie and Leung 2011; Thüm et al. 2014]. Third, effects of variations often propagate globally in programs, making it hard to detect interactions involving lots of variations even with complex data-flow analysis [Lillack et al. 2014].

To tackle these challenges, researchers have proposed dynamic analysis techniques that analyze the effects of multiple variations by efficiently tracking variations at runtime. Researchers have applied these techniques to various scenarios, such as testing highly configurable systems [Nguyen et al. 2014], understanding feature interactions [Meinicke et al. 2016] and configuration faults [Su et al. 2007], monitoring information flow of sensitive data [Austin and Flanagan 2012; Austin et al. 2013; Devriese and Piessens 2010; Kim et al. 2015; Kolbitsch et al. 2012; Kwon et al. 2016], and detecting inconsistent updates [Hosek and Cadar 2013; Maurer and Brumley 2012; Tucek et al. 2009]. These techniques are similar, and often called differently in different communities, such as variability-aware execution [Kästner et al. 2012; Meinicke et al. 2016; Nguyen et al. 2014], faceted execution [Austin and Flanagan 2012; Austin et al. 2013], coalescing execution [Sumner et al. 2011], shared execution [Kim et al. 2012], and multi-execution [De Groef et al. 2012; Devriese and Piessens 2010]. Our work is built on these ideas, and we use the name variational execution in this work, as we target primarily analyzing and testing configuration options in programs.

Previous studies have shown that variational execution can be useful in many scenarios with promising results. Since variational execution itself is not a core contribution in this work, we defer the comprehensive discussion of different applications to Section 7. Nonetheless, we highlight a few interesting results: Nguyen et al. [2014] applied variational execution to identify plugin conflicts in WordPress. Their variational execution engine can analyze 2^{50} combinations out of 50 plugins within seven minutes and found a previously unknown plugin conflict. Meinicke et al. [2016] used variational execution to understand the shape of configuration spaces in different programs and found interesting characteristics of how options interact. Austin and Flanagan [2012] and Austin et al. [2013] demonstrated usefulness of variational execution in guaranteeing non-interference of sensitive data between different confidentiality levels. Their prototype implementations in JavaScript and Jeeves can prevent cross-site scripting attacks and handle complex information flow in a conference management system. Sumner et al. [2011] showed that a form of variational execution can exploit similarities among data processing of similar inputs and gain a speedup of 2.3 without precision lost. Although variational execution has been explored before with promising results, a universal and scalable implementation is still missing. Existing implementations typically have severe scalability issues and work only with small academic examples. With a better implementation, we have a better chance of scaling existing applications and applying variational execution to broader application scenarios and more use cases.

Existing implementations rely on either *manual modification to the source code* [Austin et al. 2013; Schmitz et al. 2018, 2016] or *modification to the language interpreter* [Meinicke et al. 2016; Nguyen et al. 2014]: On the one hand, variational execution can be achieved by writing the source code to use some libraries or programming language constructs, so that the programs compute with multiple values in parallel [Austin et al. 2013; Schmitz et al. 2018, 2016]. Implementations of this kind put a heavy burden on developers because the use of these libraries or language constructs usually obscures the original programs. Moreover, rewriting existing programs is often tedious and

error-prone. On the other hand, variational execution can be achieved by executing a normal program with a special execution engine, such as an interpreter that tracks multiple values in parallel with special operational semantics for each instruction [Austin and Flanagan 2012; Meinicke et al. 2016; Nguyen et al. 2014]. Modified interpreters often suffer from a conflict between functionalities and engineering effort: It would be painstaking to modify a mature interpreter like OpenJDK, though it fully supports all functionalities of the language, whereas it takes less engineering effort to modify a research interpreter such as Java PathFinder [Havelund and Pressburger 2000], which however provides incomplete language support and often mediocre performance.

We present a new way of implementing variational execution. Our approach sidesteps *manual modification to the source code* and *brittle modification to the language interpreter*. The key idea is to automatically transform programs in their *intermediate representation*. Specifically, we transparently modify Java bytecode automatically to mirror the effects of a manual rewrite. The resulting bytecode can then be executed on an unmodified commodity JVM.

Transforming programs at the intermediate language level has several benefits. First, intermediate languages often have simple forms and strong specifications, both of which facilitate automatic transformation. Second, source code is not required, allowing us to transform also libraries used in the target programs. We can even analyze other programming languages that are compilable to the same intermediate language. Third, existing optimizations of the execution engine can be reused; in our case, our transformed bytecode can take advantage of just-in-time compilation and other optimizations provided by modern JVMs. Finally, modifications at the intermediate level remain portable. Our transformed bytecode can be executed on any JVM that implements the JVM specification.

Transformations are nontrivial and not always local. While many bytecode instructions can be transformed in isolation, encoding conditional control flow in a commodity JVM requires careful encoding, such that both branches of control-flow decisions can be executed in different configurations, before subsequent computations are merged again, to maximize sharing overall. In additional, data-flow analyses are required to handle values on the operand stack between blocks and object initialization sequences for variational execution. Finally, we perform additional optimizations to statically pinpoint instructions that do not need to be transformed, because they are guaranteed to be not related to variations in the program.

We formally prove that our transformation of control flow is correct, statically guarantee optimal sharing for a large subset of possible control-flow graphs. Additionally, we empirically evaluate performance, comparing execution time and memory consumption on seven highly configurable systems against VarexJ, a state-of-the-art variational execution implementation. The results show that our approach is 2 to 46 times faster than VarexJ, with 75 percent less memory. The performance results also indicate that our approach is efficient for analyzing highly configurable systems in practice.

We summarize our contributions as follow:

- We propose a novel strategy for variational execution using automatic bytecode transformation, without any manual modifications to the source code or to the language interpreter.
- We prove that our automatic transformation of bytecode is correct for all control-flow graphs and optimal with regard to sharing for a large subset.
- We propose further optimizations by performing data-flow analysis and using specialized data structures.
- We implement a bytecode transformation tool that covers nearly the entire instruction set of the Java language, with minor exceptions that we explain in Section 5. The transformed bytecode is portable to any implementation of the JVM specification.
- An empirical evaluation with 7 subject systems showing that our approach is up to 46 times faster while saving up to 75 percent memory when compared to the state-of-the-art. In

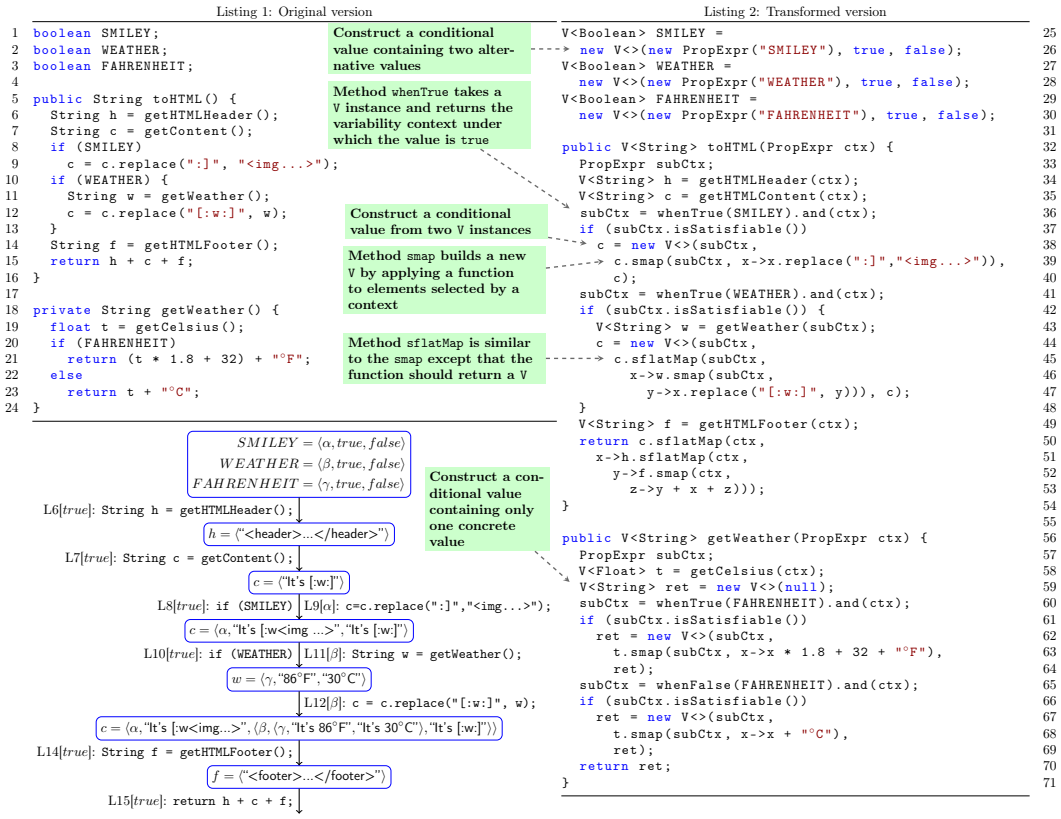


Fig. 1. Running example of this paper, modeled after WordPress [Meinicke et al. 2016]. Listing 1 shows the original source code without variational execution. Bottom left illustrates variational execution by showing the execution trace. Listing 2 hints at our variational execution transformation. The transformation is shown in Java for better readability.

addition to statically guaranteeing optimal sharing for 89.7 percent of methods, our approach achieves optimal sharing at runtime for 99.8 percent of all other method executions.

We hope that the way we transform bytecode can inspire more efficient implementation of similar techniques such as symbolic execution. Although we focus on Java bytecode in this work, we can potentially generalize the core ideas to other programming languages and other analyses, by performing a similar transformation at the well-defined intermediate representation form of existing compiler frameworks like LLVM.

2 BACKGROUND AND MOTIVATION

As necessary background for our approach, we introduce the core concepts of variational execution and show how it can be achieved with manual source-code transformation, hinting at the key ideas of our automatic bytecode transformation.

2.1 Variational Execution

There are two main concepts that distinguish variational execution from concrete execution: **conditional values** and **variability contexts**.

The key idea of variational execution is to execute a program with *concrete values*, but support *multiple alternative concrete values* for different configurations. That is, whereas each variable has one concrete value in concrete execution (e.g., $x = 1$), the concrete value of a variable may depend on the configuration in variational execution—we say the variable has a **conditional value** [Erwig and Walkingshaw 2013]. A conditional value does not store a separate value for each configuration (exponentially many), but partitions the configuration space into *partial spaces* which share the same value. That is, all configurations sharing the same concrete value are represented only once in the conditional value. Partial configuration spaces are expressed through propositional formulas over configuration options, such as $(a \vee b) \wedge \neg c$ representing the potentially large set of all configurations in which configuration options a or b are selected but not c; a tautology (denoted as `true`) describes all configurations, a contradiction (denoted as `false`) none. Conditional values are typically expressed through possibly-nested choices over formulas (or if-then-else expressions), such as $x = \langle a, \langle \neg b \vee c, 1, 3\rangle, 2\rangle$, which means: x has the value 1 in the partial space $a \wedge (\neg b \vee c)$, 3 in $a \wedge \neg(\neg b \vee c)$, and 2 in $\neg a$. With this representation, we can reason about configuration spaces with SAT solvers and BDDs.

Variational execution uses conditional values with the notion of performing a computation conditionally in a **variability context**, similar to a path condition in symbolic execution: An operation will only modify values in the part of the configuration space indicated by the current variability context (that is, we conceptually split the execution). Again, formulas over configuration options are used to express the variability context.

We assume a finite configuration space in which we know concrete values for all configuration options. Throughout this paper, we focus on boolean options when we discuss configuration options because they are common and easy to reason about with standard tools. Other options (e.g., strings, numeric values) with *finite domains* can be encoded as a set of boolean options. Using other solvers to circumvent the encoding is possible as well. The support of options with finite domains is entirely hidden behind the abstraction of conditional values, so it is orthogonal to the discussion in this paper.

Operations on conditional values can often be shared. If none of the used variables have alternative values, an instruction only needs to be executed once for all configurations (we say that we are executing under the `true` context). We begin execution in the `true` context, and only split into restricted variability contexts when configuration options influence execution—directly or indirectly. This conservative execution splitting strategy allows us to aggressively share executions that would otherwise be repeated once per configuration. This sharing avoids nonessential computations and makes variational execution efficient in many scenarios.

Comparing to Symbolic Execution. Despite some similar concepts, there are important differences between variational execution and symbolic execution. A *conditional value* in variational execution is fundamentally different from a *symbolic value* in symbolic execution, in that the former represents *a finite number of concrete values* while the latter often represents *an infinite set of possible values* of a given data type. Unlike symbolic execution where operations are carried out on symbolic values, variational execution always computes with concrete values; symbols are used only to describe configuration spaces for distinguishing alternatives and for describing contexts, but never intermix with concrete values. For this reason, loop bounds are always known concrete values in variational execution, and we avoid other undecidability problems. By considering finite configuration spaces, reasoning about configuration space of conditional values involves *inexpensive and decidable* satisfiability checks with SAT solvers or BDDs, while symbolic execution is often limited by *expensive* constraint solving and the types of theories the underlying constraint solver supports. For instance, reasoning about array elements in variational execution is fast, because we know the concrete array indexes and elements, in contrast to symbolic execution where a symbolic array index can dramatically slow down constraint solving because it can potentially refer to every element in the array.

Furthermore, variational execution has different concepts of managing state and forking and joining when compared to symbolic execution. Symbolic execution often forks new states either completely or partially at every conditional branch, often resulting into exponentially many paths in practice, commonly known as the path explosion problem. For example, Meinicke et al. [2016] has demonstrated that state-of-the-art symbolic execution implementations for Java split of separate executions on variability and share only a common prefix. Some symbolic execution engines merge state from different paths to share executions after control flow decisions, for example, introducing new symbolic values or using *if-then-else* expressions to represent differences among values from different paths—different designs make different tradeoffs with regard to performance, precision, and implementation effort [Baldoni et al. 2018; Sen et al. 2015]. Our implementation of variational execution uses a design that maximizes sharing. It maintains a single representation of all state throughout the execution where differences are represented at fine granularity (variables and fields) with conditional values. State is always modified under the current variability context, which is equivalent to merging state after every single statement. In addition, we join control flow as early as possible to avoid repeating executions, as we will discuss in Section 4.

Example of Variational Execution. As an example, consider Listing 1 in Figure 1, a simplified implementation of a blogging system modeled after WordPress. The blogging system has three variations, based on options for smiley rendering and inlining weather reports, which affect how HTML code is generated. In its current form, there is an issue: if both `SMILEY` and `WEATHER` are enabled, the replacement of a smiley image takes precedence and breaks the expansion of weather information, resulting in outputs like "[:w😊".

In order to ensure the absence of interaction bugs like this, typical testing techniques would try all combinations one by one, resulting into 8 executions of the same program in this case. Moreover, single executions alone reveal little information about the causes of interaction bugs, especially for cases where interactions of options have global effects on the execution.

Variational execution is much more efficient for detecting interactions like this. The execution trace in the bottom left of Figure 1 illustrates how variational execution explores all possible interactions among `SMILEY`, `WEATHER` and `FAHRENHEIT` in *a single run*. Boxes represent relevant program states and arrows denote execution steps. The executed statements are displayed beside arrows, together with the variability contexts. An execution trace like this can also be generated by logging and aligning concrete executions of all possible configurations, but Meinicke et al. [2018] showed that variational execution is much more efficient, sidestepping correctness and performance issues of alignment.

After marking the three `boolean` fields as options (e.g. via Java annotation), variational execution initializes them with *conditional values*, representing both `true` and `false`. The symbols α, β, γ denote the three options respectively. Variational execution runs Line 6 and Line 7 once under the *variability context* of `true`, meaning that they are shared across all configurations. Sharing like this enables variational execution to explore large configuration spaces efficiently. To highlight sharing, we put all shared statements to the left of the arrows in the execution trace. The execution is split when it comes to the first `if` statement, where `c` is modified only under the *variability context* of `SMILEY`. At this point, the content of `c` changes from *containing one value for all configurations* to *having two alternative values depending on option* `SMILEY`, and this change is reflected in the *conditional value* assigned to `c`. Finally, variational execution is able to share the execution of common code again at Line 14, after splitting executions in two `if` branches.

This example illustrates the benefits of variational execution. We can spot the problematic interaction of `SMILEY` and `WEATHER` by inspecting the conditional value of `c`, as shown in the execution trace. In fact, all possible interactions are recorded and detectable by inspecting *conditional*

values during the variational execution. All information about how options interact can be obtained after one single run of variational execution, in contrast to exponentially many with normal execution, and the difference would still not be obvious without aligning all traces of normal execution. The effectiveness of variational execution comes from using *variability context* to manage splitting and sharing of executions.

VarexJ. The state-of-the-art implementation of variational execution for Java is VarexJ [Meinicke et al. 2016]. VarexJ is implemented on top of Java PathFinder's (JPF) interpreter for bytecode [Havelund and Pressburger 2000]. For this reason, VarexJ also inherits several limitations that restrict the programs it can analyze. First, JPF is not a complete implementation of the JVM specification, so it provides incomplete support for language features, such as concurrency and native methods. Second, unlike commodity JVMs, JPF does not provide any optimizations over programs being executed, such as just-in-time compilation. Third, executing programs on JPF is slow because JPF itself is implemented as a single-process Java application. The nature of meta-circular interpreting causes a significant performance penalty. Our approach sidesteps these problems by not modifying the JVM, but transforming the Java bytecode.

2.2 A Manual Rewrite

To illustrate how variational execution can be achieved on a commodity JVM, we illustrate how the source code of our WordPress example in Figure 1 can be manually rewritten in Listing 2 of Figure 1. We show the rewrite in Java source code for better readability, as the same program in bytecode is typically longer and harder to read, potentially obscuring the essential ideas of our rewriting. This manual rewrite in Listing 2 also introduces the key ideas (highlighted as floating boxes) used later in our automated bytecode transformation. A rewrite in bytecode is also available in the appendix.

We introduce *variability contexts* in all methods, represented by instances of the `PropExpr` class, which model propositional expression over configuration options. Variables are rewritten to use a new `V` type to store *conditional values*, either a single value for all configurations or different values for different configurations. To manipulate values in `V` objects, we use `smap` and `sflatMap` methods. The `smap` method applies a function to each alternative value of a `V`, and the `sflatMap` method does the same but allows to split configuration spaces, producing more alternatives. For example, the operation `v.smap(ctx, f)` on a conditional value `v` of type `V<T>` takes as arguments (1) a variability context `ctx` and (2) a function literal `f` of type `T => U`, representing the pending operation. It returns a new V instance of type `V<U>` that results from applying the function `f` to each concrete element that exists under `ctx` in `v` (recall that a conditional value stores concrete values along with the variability contexts under which they exist). The `sflatMap` method works similarly, but takes functions of type `T => V<U>`.

Note that the manual rewrite shown in Listing 2 is not exactly the same as our bytecode transformation, but close enough to show the key ideas. An automatic rewrite of our running example in bytecode is available in the appendix for reference. We will discuss in detail how such a rewrite can be automated in bytecode in Section 3 and Section 4. Nonetheless, we can already preview a few key points from this manual rewrite:

- Variables store conditional values, represented by `V` objects.
- Most operations on conditional values (e.g., calling the `replace` method, String concatenation) are redirected with `smap` and `sflatMap` and applied to all alternative concrete values. It fact, this replacement is sufficient for most bytecode instructions, as we will see in Section 3.1.
- Both the `if` branch and the `else` branch of an `if-else` statement are transformed into an `if` statement, a statement that checks whether there exists any partial configuration under which

the surrounded code will be executed. If such a partial configuration exists, the surrounded code will be executed under a restricted variability context (e.g., Line 36–40). We will discuss transformation of control-transfer instructions in Section 4.

- All method calls have one additional parameter `ctx`, representing the variability context under which this method is called. The variability context restricts all instructions of that method invocation. Also, multiple `return` statements in the same method are replaced with temporary assignment to a local variable, which is returned in the end of the method. Transformation of method calls and method returns will be further discussed in Section 3.2.

The transformation from normal code to variational code is nontrivial and obscures the program. For example, we almost double the size of Listing 1 in order to transform a simple example into a variational execution version. The introduction of `smap` calls and complicated control-transfer structures also obscure the intention of the original program, making it hard to understand and debug. This puts a heavy burden on the developers to understand variational execution and how to use it correctly. All of these issues can be resolved if we adapt an automatic transformation approach that is transparent to developers. As we will see later in Section 5, our transformation is also able to automatically decide which parts of a program need to be transformed, as it is likely that some parts are not related to variations, such as the code before the first `if` statement (Line 7) and the code after the second `if` statement (Line 14) in Listing 1.

3 BYTECODE TRANSFORMATION

We discuss our transformation in two steps. First, in this section, we discuss how to transform all instructions that are executed in a given variability context. The transformation of control flow, which may change variability context, is nontrivial and orthogonal, so we discuss it second in Section 4. We describe transformations for similar instructions together, following the grouping of the JVM specification [Lindholm et al. 2015].

In a nutshell, we transform each bytecode instruction of the original program into a sequence of bytecode instructions. Ideally, the transformation of most instructions should be local, meaning that the transformation of the current instruction should not be affected by other instructions around it. However, this locality assumption is not generally possible because an instruction often affects another instruction by leaving data on the operand stack. The operand stack is used internally in the JVM for exchanging data between instructions. Some instructions load values (e.g., constants or values from local variables or fields) onto the operand stack, while other instructions take values from the operand stack and operate on them. Results might be pushed back onto the operand stack as a result of an operation. The operand stack is also used to prepare parameters to be passed to method invocations and to receive return values.

To assist local transformation of individual instructions, we introduce several transformation invariants:

Invariant 1 All local variables and fields store conditional values.
Invariant 2 All values on the operand stack are conditional values.
Invariant 3 All methods take conditional values as parameters and return conditional values.

We ensure that these invariants hold *before and after* the execution of each transformed bytecode sequence. They help us establish a common ground about the environment, enabling concise transformations of most instructions. In addition, we assume that each instruction is executed in a local variability context. We will explain how variability contexts are propagated and changed as part of our discussion of control flow in Section 4.

3.1 Basic Lifting

To achieve our invariants, we change all parameters and local variables in a method frame to the `V` type to store conditional values. Primitive types are boxed in the process.

Load and Store Instructions. Load and store instructions transfer values between the local variables and the operand stack. Since we assume local variables and stack values to represent conditional values (*Invariant 1*, *Invariant 2*), we can directly load them with the `aload` instruction (replacing load instructions for primitive types if needed). Store instructions require more attention, because they may be executed under a restricted variability context, in which case not all values shall be overwritten. For example, suppose we have $x = 1$ under context `true`, but store 2 to x under context `A`, then x stores the conditional value $\langle A, 2, 1\rangle$ instead of 2. To this end, we always create a new conditional value, compressing the updated values under the current context with possibly unaffected old values. As an example, consider the `V` constructor call when `c` is updated in Line 38–40 of Listing 2.

Arithmetic and Type Conversion Instructions. Arithmetic and type conversion instructions compute a result based on one or two values from the operand stack, and then push the result back on the operand stack. For example, the `iadd` instruction takes two *int* values from the stack, adds them together and pushes the result back. Given *Invariant 2*, we need to pop and push conditional values. We achieve this by invoking `smap` with the current variability context on the stack's conditional values, performing the original arithmetic or type conversion operation on each alternative concrete value. For operations on two conditional values, we combine `sflatMap` and `smap` to compute results for all possible combinations. For example, the original floating point calculation in Line 21 of Figure 1 is transformed to a `smap` call in Line 63.

Operand Stack Management Instructions. Operand stack management instructions directly manipulate entries on the operand stack, such as `pop` for discarding the top value, and `swap` for swapping the top two values. They work the same for conditional values and concrete values, and therefore do not need to be transformed. A technical subtlety in Java is that some primitive values (e.g., long, double) are represented by two 32-bit values on the stack, but only by a single reference value for a conditional value; here we adjust stack operations accordingly.

3.2 Method Invocation and Return

Method invocations pass the top stack values as arguments to the method and push the method's result back to the stack. Non-static methods also take their receiver from the stack. Since method arguments and return types are conditional values, just as stack values (*Invariant 2*, *Invariant 3*), they can be passed along directly. If a method call has multiple receiver objects, we call the method for each of them in the corresponding variability context and merge results using a `sflatMap` call.

Special handling is required though in cases in which *Invariant 3* does not hold for the target. Ideally, all classes and all methods in variational execution should be transformed, but this is not always possible in practice because of the environment barrier. At some point, variational programs may need to interact with an environment that does not know about variational values and variability contexts. The environment barrier can be at different places, depending on how the system is implemented (e.g., between user code and library code, between Java code and native code, between the program and the operating system or network), but can never be avoided entirely. When hitting the environment barrier, we have three options:

Multiple invocations. For *side effect free* methods, we can invoke the target method multiple times for each feasible combinations of concrete argument values, merging the results into a single conditional value. Since the method is side effect free, invoking it repeatedly with different arguments does not change the program states, it just forgoes potential sharing.

Model classes. We can always provide variational models for the environment, for example, replacing all reads and writes to a file with a special implementation that can store alternative file context under different contexts. Such model classes are common in model checking and symbolic execution [d'Amorim et al. 2008; Sen et al. 2015; von Rhein et al. 2011] and have been explored in variants of variational execution for database storage [Yang et al. 2016]. Model classes can also be used to provide more efficient variational implementations for classes than would be achieved with our automated transformation, as we will discuss in Section 5.

Abort. Finally, we can execute the program but abort execution when we reach the environment barrier at runtime. This way, we can still support executions that do not cross the barrier, even though the source code refers to nonvariational methods. Furthermore, we can allow calls to nonvariational methods during the execution when they are shared by all configurations (with variability context *true*) in which all parameters have only a single concrete value.

In our approach, we transform all methods possible, including libraries, to push the environment barrier as far outside as possible. In the JVM, the environment barrier often manifests as native methods, i.e., methods that are hard-coded in the JVM in other programming languages such as C and C++. We maintain a list of model classes and side-effect free methods that are automatically applied when encountered. For all remaining calls to nonvariational code, we issue warnings during transformation and abort the execution at runtime when invoked. We then manually and incrementally inspect aborts in our executions and mark methods as side-effect free or develop model classes as needed. In fact, so far, we needed to implement model classes only for a small number of classes. We have not yet encountered executions that heavily rely on variational interactions with the environment and thus require additional model classes.

Return instructions are more straightforward to transform than method invocation instructions. To not prematurely end the execution of a method at a return instruction, we rewrite the method to use a single return instruction at the end of the method. If the method being transformed has more than one return instructions, we rewrite all of them to jump to a single return at the end of the transformed method. If necessary, we store the values of different original return instructions in a variable. Technically, we again replace all non-void returns by a single `areturn` instruction, returning a reference to the resulting conditional value. For example, see how Line 21 and 23 are transformed to Lines 62, 67 and 70 in Figure 1.

3.3 Using Objects

In the JVM, both *class instances* and *arrays* are objects, but the JVM creates and manipulates class instances and arrays using distinct sets of instructions. This section presents our transformation of them respectively.

Class Instances. We transform all fields of a class instance to have the conditional value type. The key idea is to maximize sharing of data across similar class instances. If two instances of the same class only differ in one field, we represent the difference in a conditional value for that field, rather than as a conditional reference to two copies of the object. This design stores variability as local as possible to avoid redundancy in memory and in computations [Meinicke et al. 2016]. As fields store conditional values (*Invariant 1*), reads and writes to fields work just as loads and stores to local variables.

A technical challenge to independent transformation of bytecode instructions arises for the `new` instruction used to instantiate classes and push them to the operand stack. The challenge is that the `new` instruction creates an uninitialized object that cannot be passed as a reference for safety reasons until the object's constructor is invoked on it, and thus cannot be wrapped in a `V` type as needed for *Invariant 2*. Instead, we treat `new` and the subsequent initialization sequence

as *one bytecode instruction* for our transformation. Whenever we encounter a `new` instruction, we use a data-flow analysis to identify the relevant following initialization sequence, re-arranging the original bytecode if necessary to separate object initialization from other instructions (e.g., instructions to compute constructor parameters).

Arrays. For a given array, we transform it into an array of conditional values to again store variability as local as possible to preserve sharing. To support arrays of different length though and fulfill our invariants, we support also variations of arrays. That is, an array of objects (`Object[]`) would be represented as a conditional array of conditional objects (`V<V<Object>[]>`). Type erasure in Java complicates the implementation, but this can be solved by inserting additional dynamic type checks.

We arrived at this design after considering several tradeoffs: Our representation can store variability more locally, avoiding that a single variation in an entry requires to copy the entire array; also load and store operations are simple and fast. Overheads are only encountered for arrays with different length in different configurations, which is less common than variability in values in our experience. An alternative design could loosen our invariants for arrays and create a single maximum-length array of conditional values (based on the length of the configuration with the longest array; `V<Object>[]`) and a shadow variable and extra instructions for bookkeeping and length checking, but we only expect marginal performance benefits from this more complicated design.

4 CONTROL TRANSFER

After describing how to transform bytecode instructions within a given variability context, we now focus on how to transform control-flow related constructions that may change variability contexts by splitting or joining executions. For example, in a branching statement the condition may differ among configurations, such that we may need to execute both branches under corresponding variability contexts, but join afterward to maximally share subsequent executions.

We significantly change the way programs are executed and track and change variability contexts. As introduced in Section 2, variability contexts are propositional formulas over configuration options that describe the partial configuration space for which an instruction is executed, similar to path conditions in symbolic execution. Instructions executed in a variability context only have an effect on the state of that partial configuration space, as discussed, for example, for store instructions in Section 3.1. The challenge is now to propagate and change variability context to achieve a shared execution for all configurations with maximal sharing.

In this section, we explain how we structure the program in blocks with the same variability context, and how we transfer control and contexts among these blocks. Subsequently, we then discuss two important properties of our design: (1) that variational execution preserves behavior of the original program (*Correct Execution Property*) and (2) that control transfer among blocks is efficient (*Optimal Sharing Property*). Finally, we present some technical challenges and their solutions regarding stack values during control transfer.

4.1 VBlock

We group all instructions that are statically guaranteed to always share the same variability context at runtime in a *VBlock*. VBlocks are separated by *conditional* jumps, that is, jumps that may depend on conditional values, in which case we may "split" the execution. After executing multiple VBlocks we may "join" the execution in another VBlock with a broader variability context (such joining is rare in symbolic execution approaches). For example, String replacement of a smiley image (Line 9) in Listing 1 has a more restricted context than the `getHTMLHeader` call (Line 6) because Line 9 is only executed when `SMILEY` is `true`, whereas the later `getHTMLFooter` call is again shared among all configurations.

VBlocks are similar to basic blocks in traditional program analyses. However, unlike basic blocks, which group individual instructions together because they are always executed in sequence, VBlocks group basic blocks together because they always share the same variability context. Thus, there can be jumps inside a VBlock as long as they do not depend on conditional values and thus share the same variability context.

Bytecode instructions can be partitioned into VBlocks by merging basic blocks in a control-flow graph iteratively until a fixpoint is reached. A block B_1 can be merged with a successor B_2 if the jump between B_1 and B_2 is not conditional (e.g., `goto` or *if* statement with non-conditional expression)[1] and all predecessors of B_2 are in the same VBlock. The latter condition is needed to recognize potential join points, when a block can be reached from two different VBlocks. Hence, a VBlock can be terminated by either a conditional jump or an unconditional jump. A VBlock can end with an unconditional jump if, for example, while merging basic blocks to form VBlocks, basic block `A` has an unconditional jump to basic block `C`, while basic block `B` has a conditional jump to `C`. We cannot merge `A` and `C` into one VBlock because of the conditional jump from `B`. Thus, `A` and `C` have to be separated into two different VBlocks with an unconditional jump between them.

4.2 Execution Strategy

This subsection presents how VBlocks are used. We first outline the goals of using VBlocks to achieve splitting and joining execution. Then, we present a solution that achieves our goals and provide an example.

Goals. Whereas variational-execution approaches that modify interpreters (such as VarexJ) can track multiple instruction pointers and their variability contexts, we need to cope with the fact that the instruction pointer of an unmodified JVM can only point to a single location at a time. So instead of changing the control transfer mechanism of the JVM, we use VBlocks to organize and create the execution order we want. At a high level, we pursue the following:

- Both branches of a conditional jump can be executed under corresponding restricted contexts (we call them "subcontexts"). That is, we are able to split execution.
- The code after both branches of a conditional jump should be executed only once for mutually exclusive contexts. That is, we should join execution as early as possible.

Context propagation. Using VBlocks, we modify control flow decisions and manipulate variability contexts to achieve splitting and joining. The key idea of our design is to associate each VBlock with a variability context (a fresh local variable). We dynamically update variability contexts along execution to keep track of which VBlock(s) can be executed next and under which context. At any point in a method's execution, all VBlocks with a *satisfiable* variability context (i.e., the proposition formula is satisfiable) can be executed. The order in which multiple VBlocks with satisfiable contexts are executed does not matter for correctness, but does matter for performance, as we will show in Section 4.3.

At a jump between VBlocks, we transfer the current block's variability context to the target block's context. If the jump is conditional, we split the current variability context and transfer the two mutually exclusive contexts to the two successor VBlocks of the jump. The split is determined by the partial configuration space in which the *if* statement's expression evaluates to true.

To describe the control transfer more precisely, let us denote the sequence of VBlocks as $b_0, b_1, \ldots, b_n (n \geq 0)$, where b_0 represents the entry node in the control flow graph and b_n represents

[1] *Invariant 2* implies that all values evaluated in an *if* statement are conditional, however, as we will discuss later in Section 5, we can optimize the transformation to statically recognize values that will not depend on configuration options, including, in the simplest case, constants.

the exit node. Let us denote the variability context of a VBlock b_i as $\phi(b_i)$ (stored in a fresh local variable for each VBlock).

- At the beginning of a method execution, we initialize $\phi(b_0)$ with the method context, and $\phi(b_i) = \mathit{False}$ for all other VBlocks to indicate that only the initial VBlock of the method can be executed.
- After executing a VBlock b_i, we remember its variability context $\Phi = \phi(b_i)$ and then set that variability to *False*, indicating that this block should not be immediately executed again. We subsequently propagate its prior variability context Φ as follows:

(1) If the execution of VBlock b_i ends with an unconditional jump (e.g., `goto` instruction) to another VBlock b_j, the context of b_j is updated as a disjunction between the current context of b_j and b_i's prior context Φ. A disjunction is required because the target block may already have been executable under a different context, which we now broaden to join executions.

$$\phi'(b_j) = \phi(b_j) \vee \Phi \tag{1}$$

(2) If the execution of VBlock b_i ends with a conditional jump with two possible target VBlocks b_j and b_k,[2] we split the execution based on the condition of the jump (usually the top value on the stack representing result of evaluating an *if* statement's expression). Let us denote the variability context in which the jump condition indicates a jump to b_j as X. For example, the condition of the first *if* statement in our WordPress example is $\langle \mathit{SMILEY}, 1, 0 \rangle$, which indicates the *then* branch should be taken under context $X = \mathit{SMILEY}$. We update the variability contexts of b_j and b_k as follows, again considering potential joins:

$$\phi'(b_j) = \phi(b_j) \vee (X \wedge \Phi) \qquad \phi'(b_k) = \phi(b_k) \vee (\neg X \wedge \Phi) \tag{2}$$

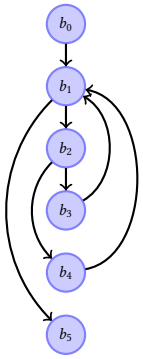
- After propagating the variability context, the control transfer (i.e., the actual instruction pointer in the JVM) does not actually follow the jump.

Execution Order. The actual execution order though (in terms of moving the instruction pointer) is independent from the transfer of variability contexts. We start execution at the beginning of the method with b_0. At the end of a VBlock b_i, we jump to the next VBlock b_{i+1} by default, even if the block ended with a different jump. If that VBlock's variability context is unsatisfiable, we proceed to the next VBlock, and so forth. We only jump back to a VBlock with a lower index (using a plain `goto` instruction) when we update the variability context of an earlier block to be satisfiable as part of the described context transfer. This way, the instruction pointer is always at an unsatisfiable block (to be skipped) or at the satisfiable block with the lowest index. This strategy ensures that later VBlocks are always executed with joined variability contexts from earlier VBlocks and that VBlock b_n is executed last with the full method context. For that reason, the indexing order of VBlocks matters. Figure 2 illustrates the idea of jumping among VBlocks with a concrete example.

Ordering VBlock Execution. Given that we always execute the first VBlock with a satisfiable variability context and always join at later VBlocks, we can execute the same method in different ways by changing the way we order the VBlocks. We can reorder VBlocks in different orders as long as the first and last VBlock remain constant (the last block ending with a return statement must be executed last) and always achieve equivalent (i.e., correct) results, as we will show in Section 4.3. However, as the block order determines the join points, different orders may be more or less effective at joining early and sharing subsequent computations.

To maximize sharing during the execution (i.e., prefer executing a block once under a broader variability context rather than multiple times under narrow contexts), we order VBlocks based on

[2]We transform switch statements into an equivalent series of if-else statements to simplify our design of control transfer.



b_0	`int i = 0;`	Update: $\phi(b_1)$	Goto b_1
		Skip to b_2 unless $SAT(\phi(b_1))$	
b_1	`while (i < b) {`	Update: $\phi(b_2)$, $\phi(b_5)$	Goto b_2
		Skip to b_3 unless $SAT(\phi(b_2))$	
b_2	`i++` `if (i != a)`	Update: $\phi(b_3)$, $\phi(b_4)$	Goto b_3
		Skip to b_4 unless $SAT(\phi(b_3))$	
b_3	`continue;`	Update: $\phi(b_1)$	Goto b_1
		Skip to b_5 unless $SAT(\phi(b_4))$	
b_4	`...` `}`	Update: $\phi(b_1)$	Goto b_1
b_5	`return;`		

Fig. 2. An example illustrating control-flow encoding through updates of variability contexts and jumps between blocks.

the *strict transitive predecessor* relation in the control-flow graph. A VBlock b_i is a strict transitive predecessor of b_j if there is a path from b_i to b_j in control-flow graph, but not from b_j to b_i (i.e., not in a loop). *For any pair of VBlocks, if one VBlock is a strict transitive predecessor of the other, the transitive predecessor shall have the lower VBlock index to be executed first.* For other pairs, we preserve the original lexical order produced by the compiler as a default.

In the next subsection, we will show that the above partial order is sufficient to statically guarantee optimal sharing on a subset of control-flow graphs, regardless of the original lexical order of the bytecode, but that optimality cannot be statically guaranteed for all control-flow graphs. We will also experimentally show in Section 6 that this order is nearly always optimal for the remaining control-flow graphs.

Example. Let us exemplify our solution by stepping through the `getWeather` method in Listing 2. There are four VBlocks: code before the `if` statement (b_0, Line 57-59), then branch (b_1, Line 60-64), else branch (b_2, Line 65-69) and return block (b_3, Line 70). These blocks are already indexed according to the strict transitive predecessor relation: b_0 is executed first, b_1 and b_2 are executed before b_3; the order between b_1 and b_2 is merely derived from the lexical order and could be switched. Initially, $\phi(b_0) = MCtx$ (method context) and $\phi(b_1) = \phi(b_2) = \phi(b_3) = False$. After executing b_0 at Line 59, $\phi(b_1)$ and $\phi(b_2)$ are updated to $\phi(b_1) = False \vee (FAHRENHEIT \wedge MCtx)$ and $\phi(b_2) = False \vee (\neg FAHRENHEIT \wedge MCtx)$, thus *splitting* the execution. Note that this update of contexts is not shown in Listing 2 because we transform the control flow in bytecode differently from how we show for Java. At this point, both $\phi(b_1)$ and $\phi(b_2)$ are satisfiable and execution continues with the next VBlock b_1. After executing b_1 at Line 64, $\phi(b_3)$ is updated to $\phi(b_3) = False \vee (FAHRENHEIT \wedge MCtx)$ because b_3 is the sole successor of b_1 in the control flow graph. We execute the next satisfiable block, which is b_2, after which $\phi(b_3)$ is updated to $\phi(b_3) = (\neg FAHRENHEIT \wedge MCtx) \vee (FAHRENHEIT \wedge MCtx) = MCtx$; thus, b_3 at Line 70 is executed last under the *joined* context $MCtx$.

4.3 Properties

We have presented how we choose VBlocks for execution. While splitting executions, we need to ensure that the execution order is correct. By always executing the satisfiable VBlock with the lowest index first and ordering VBlocks deliberately, we make sure that the joining happens as early as possible. This section formalizes these properties.

Correctness. The following property ensures that our variational execution is correct, in a sense that it preserves the semantics of the original program.

Property (Correct Execution Property). *At any point of execution, if there are multiple VBlocks with satisfiable contexts, the order in which they are executed does not affect correctness of execution.*

To prove this, we first introduce a useful lemma:

Lemma (Disjoint Context Lemma). *At any point of variational execution, the context of two different VBlocks are mutually exclusive. That is, $\phi(b_i) \wedge \phi(b_j) = False$ for any $i \neq j$.*

Mutual exclusion is guaranteed by the way we propagate contexts in Equations 1 and 2. A proof by induction can be found in the appendix. With this lemma, we prove our *Correct Execution Property* as follows:

Proof. Ensuring that VBlocks have mutually exclusive variability contexts guarantees that each VBlock operates on mutually exclusive runtime states. As we have discussed in Section 3, states (e.g., local variables, fields) are stored separately for different contexts using conditional values. Our variability contexts further ensure that all state changes only update values in the (disjoint) contexts referred to by the current variability context. Thus, execution order among satisfiable VBlocks does not affect correctness of overall variational execution. □

Optimal sharing. The main utility of variational execution is its ability to share common computations; our execution scheme pursues to perform executions with the broadest variability context possible. While we cannot share repeated executions under the same context, we can avoid executing the same VBlock under mutually exclusive contexts and rather execute it once, shared, under a broader context. In a nutshell, what we want to achieve is to execute every VBlock as few times as possible by sharing the execution of VBlocks in different contexts. This sharing is crucial for the overall performance of variational execution, otherwise it may degrade to executing each variation in a brute-force way or sharing only common prefixes of traces, conceptually equivalent to joining only after the very last instruction.

In order to formalize *optimal sharing*, we define a variational trace as a chronological sequence of VBlocks executed during variational execution. We denote a variational trace as a sequence of executed VBlocks with corresponding variability context, e.g., $t_v = [b_0^{True}, b_1^{\alpha}, b_2^{\neg\alpha}, b_3^{True}]$. Conceptually, a variational trace corresponds to a separate concrete trace for each configuration, in our example $t_\alpha = [b_0, b_1, b_3]$ and $t_{\neg\alpha}=[b_0, b_2, b_3]$. Another variational execution trace that represents the same concrete traces could be $t'_v = [b_0^T, b_1^{\alpha}, b_3^{\alpha}, b_2^{\neg\alpha}, b_3^{\neg\alpha}]$. It is likely that t_v is more efficient than t'_v because b_3 is executed twice in t'_v.

A variational trace can be seen as the result of aligning multiple concrete traces. Different aligning schemes produce different variational traces (e.g., t_v and t'_v). Given a set of concrete traces, we can use sequence alignment algorithms (e.g., Needleman-Wunsch algorithm [Needleman and Wunsch 1970]) to obtain a globally optimal solution of merging concrete traces. For example, two optimal matchings of t_α and $t_{\neg\alpha}$ are $t_o = [b_0, b_1, b_2, b_3]$ and $[b_0, b_2, b_1, b_3]$. We use $length(t)$ to denote the number of elements in a trace. For example, $length(t_v) = 4$, and $length(t'_v) = 5$.

Definition 4.1 (Optimal Sharing). Given a variational trace t_v and its corresponding set of concrete traces $t_1, t_2, \ldots, t_m$, we say t_v has optimal sharing if and only if $length(t_v) = length(t_o)$, where t_o is the optimal matching of $t_1, t_2, \ldots, t_m$.

It would be ideal if optimal sharing could be achieved for all possible programs in the wild, but there is no join strategy that could *statically* order blocks to guarantee optimal sharing for all executions of all programs. Figure 3 illustrates an example: In order to achieve optimal sharing (with optimal defined as the optimal trace alignment in the figure), there is both a case where b_3

needs to be executed before b_4 and a case where b_4 needs to be executed before b_3 after a control-flow decision at b_2 (critical nodes highlighted in the trace). That is, we cannot *statically* decide an ordering between b_3 and b_4, and even an optimal decision at runtime would have to depend on knowing the *future* execution trace. We could apply some greedy strategies to approximate optimality, but that the required runtime monitoring is unlikely to justify the performance benefits of additional sharing execution.

Fortunately, we can prove optimal sharing for static VBlock ordering for many shapes of control-flow graphs and will show in our empirical evaluation that the remaining ones (often with nontrivial interleaving of looping and branching instructions) are often optimal for actual executions.

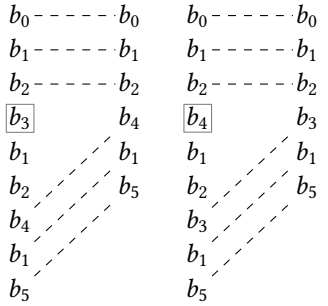


Fig. 3. An example where static order between VBlocks cannot not achieve optimal sharing. The control-flow graph is shown in Figure 2

Property (Optimal Sharing Property). *Given a control flow graph where each node represents a VBlock, our variational execution based on the strict transitive predecessor relation on this graph has optimal sharing if it is acyclic or only contains simple loops. A loop is a simple loop if it satisfies the following three criteria: (1) has only one loop header; (2) has only one exiting node; (3) has no conditional jumps among nodes in the loop.*

The proof can be found in the appendix. Intuitively, we prove by case analysis that our variational trace has the same length as the optimal alignment of corresponding concrete traces in every possible case. Since we only consider simple control-flow graphs, the length of our variational trace and the length of the optimal alignment trace can be determined from the structure of the control-flow graph.

4.4 Values on the Stack between VBlocks

In Java, blocks can leave values on the operand stack to be consumed by subsequent blocks. Since, in variational execution, there might be multiple successor blocks that will be executed, and successor blocks may not be executed immediately after their predecessor, sharing values on the stack becomes tricky. Since the operand stack in a commodity JVM is not variational itself, we cannot pop the same value from the stack under different variability contexts as possible when modifying the interpreter itself (e.g., done in VarexJ [Meinicke et al. 2016]).

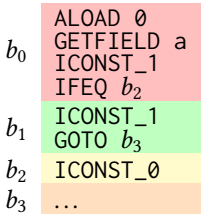


Fig. 4. A snippet of bytecode showing the scenario where VBlocks could leave some values on the operand stack after execution.

Figure 4 shows a concrete example in which VBlock b_0 leaves some value on the operand stack after execution, that both blocks b_1 and b_2 try to read. Conversely, those two blocks each leaves a (different) value on the stack that b_3 attempts to consume.

To make the transition between VBlocks possible in all cases, we need one more invariant:

Invariant 4 A VBlock does not leave any values on the operand stack at the VBlock boundary.

To meet this invariant, we store all remaining values on the operand stack (if they exist) to local variables, at the end of each VBlock. Then at the beginning of each VBlock, we check if the current VBlock expects some values from the operand stack, and load those values from corresponding local variables if so. Since we support loading and storing under different variability contexts, as discussed above, this solution generalizes to all control-flow graphs.

5 IMPLEMENTATIONS, OPTIMIZATIONS, LIMITATIONS

We implemented a bytecode transformation tool for the ideas discussed in Section 3 and Section 4, and we call it *VarexC*. We use the ASM[3] library to implement our data flow analysis and transformation of bytecode. Transformations happen at class loading time via our own class loader that transforms classes before they are actually loaded. We also save the previously transformed classes and reuse them if there are no changes. To ensure correct implementation of variational execution, we apply differential testing to compare execution results and execution traces in variational execution against brute-force concrete executions for our subject systems [Kästner 2017]. Our implementation is available on GitHub.[4] We implement transformations for all bytecode instructions and provide a mechanism for model classes, as discussed in Section 3.2. In addition to implementing the full transformation described previously, we explored two optimizations and briefly outline them. Finally, we discuss the current limitations of our tool.

5.1 Optimization: Deciding What to Transform

Not all bytecode instructions in a program may depend on configuration options. If we can statically guarantee that parts of the program never depend on conditional values or conditional jumps, we can reduce our transformation to relevant parts; this reduces the overhead of computing with conditional values and conditional jumps where not needed. Guaranteed non-conditional computations happen often in the beginning of methods and typically involve initialization sequences or constants, such as in logging statements "`System.out.println("done");`".

We designed a simple data-flow analysis to decide which instructions need to be transformed. Along the lines of a standard taint analysis, we mark all local variables and values on the operand stack as conditional or unconditional with a 'lift' bit, marking them as conditional when instructions based on other conditional values write to them.

So far, we implemented an intra-procedural analysis that assumes all fields (including fields representing configuration options, as in our WordPress example) and method parameters and method results are conditional. As such, all stack values produced by field reads, loads of method parameters, and results from method invocations are marked with the lift bit. We then propagate the lift bit to all values resulting from computations in which operands had the lift bit and to local variables when such values are stored. Based on the lift bit, we decide which control-flow decisions are conditional (i.e., potentially depend on conditional values) and compute VBlocks correspondingly. Finally, we determine with a simple control-flow analysis, which VBlocks are guaranteed to be executed with the method's variability context (in a nutshell, all VBlocks that dominate the method exit) and mark all variables stored in other VBlocks as conditional, as they may be stored only in restricted contexts. As is common for data-flow analyses, we repeat these computations until a fixpoint is reached.

Based on our analysis, we transform bytecode based on a potentially smaller number of VBlocks (because some jumps are statically guaranteed to be non-conditional when their expression does not have the lift bit). We also transform only variables and instructions with the lift bit. We introduce additional instructions to translate concrete values into conditional values when values flow from unmodified into transformed code (i.e., just wrapping the concrete value in a `V` instance, boxing primitive types if necessary), such that our invariants still hold from the perspective of the transformed instructions.

Our current analysis is very conservative, because it assumes all fields and method signatures are conditional, thus most savings relate to constants and initialization sequences. Nonetheless, in

[3] http://asm.ow2.org/
[4] https://github.com/chupanw/vbc

the programs of our evaluation (Sec. 6), we can statically decide to not lift up to 32.6 percent of all instructions, which however has only a marginal impact on performance. We hope that future work can push this analysis even further by performing an inter-procedural analysis to determine which methods and method arguments need to be transformed, potentially providing multiple transformed or partially transformed copies of the same method.

5.2 Optimization: Using Model Classes

As discussed in Section 3.2, we provide a mechanism for model classes with which we can implement custom implementations for classes where automated transformations are not possible (e.g., native methods, environment barrier) or inefficient. In fact, it is often possible to provide more efficient implementations of common data-structure implementations that are specifically designed for variability [Meng et al. 2017; Walkingshaw et al. 2014]. Our model-class mechanism allows drop-in replacements for such classes.

Variational data structures. We implemented a small number of custom variational data structures for commonly used collections. For example, instead of an automated transformation of the `java.util.LinkedList` class, which would support conditional values and conditional successors of linked-list nodes, we use a custom implementation that internally stores a list of optional elements and provides corresponding accessor functions. Similarly, rather than automated transformation of `java.util.HashSet` objects, we can represent variational sets as a mapping from values to variability contexts that describe the configuration space in which the set contains that value. As explored by Walkingshaw et al., such tailored representations are often (though not generally) much more efficient, especially when they hold many optional elements with different conditions [Meng et al. 2017; Walkingshaw et al. 2014].

Depending on different computations in different programs, the effectiveness of model classes varies. For example, our model `LinkedList` is optimized for iterating elements, so programs that iterate lists of optional entries frequently gain more benefits from our model classes. Our evaluation shows different levels of improvement after the drop-in replacement of some model classes, with up to 6 times speedup for GPL.

Custom access patterns. While custom data structures can store conditional entries more efficiently, common accessor patterns to iterate over list entries can still be very inefficient. For example, getting first the first, then the second element of a list with optional entries $[1^{\alpha}, 2^{\beta}, 3^{\gamma}, 4, 5]$ would create large conditional values (e.g., $\langle \alpha, 1, \langle \beta, 2, \langle \gamma, 3, 4 \rangle \rangle \rangle$ for the first element).

Instead, we detect common access patterns and transform them more intelligently. Instead of iterating over all elements of the list one by one (where each element can be a conditional value), we iterate over all optional elements, where the element is a concrete value, but the iteration is executed under a restricted variability context based on that element's condition, which marks under what context this element exists in the list. We integrate the most recent detection and rewrite of such access patterns of Lazarek [2017]. Our current implementation detects loops that use `iterator` and automatically transform such loops to use our specialized list more efficiently.

In our evaluation, there were only few instances that benefited from this optimization, but if they did, the improvements were substantial. For example, in *CheckStyle* (see Sec. 6), the program iterates over a list of 135 optional checks. The basic transformation results in exponential behavior, that makes it infeasible to execute the code without manual rewrites of the CheckStyle implementation, whereas our optimization of access patterns allows to execute this code fragments efficiently. Overall, several researchers have explored variational data structures and access patterns recently [Meng et al. 2017; Walkingshaw et al. 2014]. Model classes and additional rewrites during the transformation

allow us to easily integrate such advances to improve performance of variational execution on real-world systems.

5.3 Limitations

Our current implementation of variational execution has some limitations, most of which are related to low-level details of the JVM or restrictions posed by the Java runtime (e.g., we cannot directly modify classes in the `java.lang` package for safety reasons). Most limitations are engineering challenges that can be overcome with additional implementations, typically in the form of model classes.

Exception. We distinguish two types of exceptions: non-variational exceptions and variational exceptions. *Non-variational exceptions* are thrown or propagated under the current method context. *Variational exceptions* are thrown or propagated under a smaller context than the current method context (i.e., only in some partial configurations). Semantically, non-variational exceptions represent cases where invoking a method under `ctx` would always result in the same exception under *all configurations* of `ctx`, whereas variational exceptions occur *only in some partial configurations*.

Non-variational exceptions are easy to support because the control flow is the same for all configurations. In fact, we only found non-variational exceptions when executing our subject systems in evaluation.

Variational exceptions are trickier to handle because method execution might be interrupted under some partial configurations. If the exception is not caught inside method invocation, returning from a method results into a normal return value in some configurations and an uncaught exception in other configurations. Although it is possible to support variational exceptions by delaying throwing them and wrapping them together with normal return values as a conditional value, the transformation would complicate the control flow of transformed bytecode in a nontrivial way, especially if exceptions are supposed to be caught inside the current method or some outer methods. Since variational exceptions are not that common in our experience, we adopt a less efficient but easier approach to support variational exceptions: The key idea is to throw an exception immediately when it occurs and continue the rest of the variational execution only under the variability context of the exception; then we restart variational execution under the remaining contexts that did not result in the previous exception, and keep repeating until all contexts have been explored. Re-executions might affect overall efficiencies of variational execution, but we only observed variational exceptions in our own artificial examples.

Model classes. We only implemented a handful of model classes (9 classes and in total 1030 lines of Java code) to tackle the environment barrier required by our subject programs. We consider all classes that have native methods and classes that are closely related to internals of the JVM to be behind the environment barrier and use the strategies discusses above, including repeated invocations and model classes.

Currently we support a large set of Java programs, but we may need to provide more model classes if another program uses certain advanced language features. We adopt an incremental approach, in which we carefully monitor the need for model classes at runtime (i.e., when conditional values are passed across environment barriers). When implementing model classes, our main focus is to support conditional values. Symbolic execution and model checking face a similar challenge, but we argue that the implementation effort is lower for variational execution, because we compute with concrete values and can therefore delegate to existing implementations rather than reimplement abstractions of those operations.

Reflection. Reflection is relatively simple to handle due to the dynamic nature of our approach. Since reflection cannot modify bytecode (i.e., we cannot introduce conditional instructions at runtime), it does not affect our bytecode transformation of classes. We intercept reflection calls

and replace them with our special call stubs, where we wrap arguments into conditional values, append variability context to the argument list, and invoke the transformed method, just as we would transform bytecode statically. We have implemented partial support for reflection as needed by our subject systems incrementally.

Synchronization. Two instructions (`monitorenter` and `monitorexit`) are used to synchronize concurrent operations. We currently keep them as is, which implies that we lock sections for all configurations, not just in the current variability context; this may lead to over-synchronization and potential liveness issues.

Array. As we will see in Section 6, array operations are generally expensive in their current form. Especially when crossing the environment barrier, we may need to translate conditional arrays into plain concrete arrays, which can be relatively expensive if the arrays are large. A more efficient implementation of variational arrays would be future work.

Comparing to VarexJ. Our approach comes with its own limitations, but most of them can be improved with additional engineering effort. We sidestep most bottlenecks of the state-of-the-art approach (VarexJ) by transforming bytecode instead of modifying the underlying JVM. Although the limitations of VarexJ can potentially also be removed by more engineering effort, we argue that the effort in VarexJ is much higher because of those additional complexities from the JVM itself. As an example, native methods are notoriously difficult to support in Java PathFinder (JPF), the underlying JVM of VarexJ, largely because JPF has its own memory model for objects, which cannot be passed to native methods directly and therefore require additional conversion. In contrast, native methods can be supported in our approach by providing simple model classes to handle conditional values so that concrete values can be passed to native methods.

6 EMPIRICAL EVALUATION

In an empirical evaluation, we now execute a number of configurable systems to assess performance (time and memory consumption) and effectiveness of sharing. Specifically, we compare our implementation against repeatedly executing the unmodified code in all configurations (brute-force execution)[5] and against VarexJ [Meinicke et al. 2016], a state-of-the-art variational execution engine for Java, which executes bytecode with a modified virtual machine based on Java Pathfinder. While performance measures implicitly indicate the benefits of sharing, we additionally empirically assess how often our VBlock ordering results in optimal sharing at runtime, especially for methods for which we cannot guarantee optimal sharing statically.

6.1 Experimental Setup

Benchmarks. Table 1 shows the benchmark programs used in this study. For comparability, we use the same set of benchmark programs from VarexJ [Meinicke et al. 2016], which includes programs from various domains: Jetty 7 is a HTTP server; Checkstyle is a static coding style checker for Java programs; Prevayler is an in-memory database system; QuEval is an academic evaluation framework for database index structures; Elevator, GPL and E-Mail are commonly used benchmarks from the software product-line community that are designed to have many variations. These programs have 6 to 141 options, each of which is a `boolean` controlling inclusion or exclusion of a feature. Feature combinations are usually restricted by a feature model [Schroeter et al. 2012]. The goal of analyzing these programs is to estimate the effort of exploring a big configuration space, which can be useful for testing, static analysis, and so forth. We execute each program with a representative input, which

[5]For benchmark programs that have more than 20 configuration options, we randomly select 1 million valid configurations for measuring.

in each system covers all configuration options and significant parts of implementation. For example, we feed Checkstyle with 4 Java source code files and use 135 different checkers to check coding style.

Implementation and Hardware. To compare with our tool *VarexC*, we use the latest VarexJ code base as of 12/12/2017, which includes the most recent optimizations, added after the last publication. Both VarexC and VarexJ are executed with Java HotSpot™ 64-bit Server VM (v1.8.0_161). We use a laptop with 2.30GHz Intel Core i7 CPU and 16GB system memory. All results are measured when the machine is idle and unloaded.

Performance Measurement. We measure performance in three different settings:

- First, we measure the performance of executing the unmodified program in every single configuration separately on a commodity JVM. Since the execution time may differ significantly between configurations, we report both the average execution time (reported as μJVM) and the execution time of the slowest configuration (reported as max JVM).
- Second, we measure the time it takes VarexJ, the state of the art variational interpreter built on top of Java Pathfinder, to execute the program across all configurations (reported as *VarexJ*).
- Finally, we measure how long it takes to execute the program across all configurations by executing the modified bytecode with a commodity JVM (reported as *VarexC*).

Ideally, the performance of variational execution (*VarexJ* and *VarexC*) would be between the execution time of the slowest configuration (max JVM) and the combined execution time of all configurations (μJVM $\cdot$ number of configurations): Variational execution needs to at least execute all instructions of the slowest configuration, but it can usually share effort among multiple configurations.

In all three cases, we measure steady-state performance for each benchmark, based on repeated executions [Georges et al. 2007]. Steady-state measurement excludes JVM startup time, which typically dominates by JIT compilation and class loading. We do not compare startup performance because VarexJ is implemented as a Java interpreter itself—in addition to loading classes of benchmark programs, VarexJ needs to load a lot of necessary classes for the meta-circular interpreter to work, which would bias our results against VarexJ. For VarexC, we exclude the bytecode transformation time from measurement because transformation happens once for each program, similar to compiling source code. We only measure VarexC with *all optimizations* (see Section 5) for brevity. Following the suggestion from Georges et al. [2007], we measure steady-state performance in the following steps:

(1) Start a JVM invocation i and iterate the benchmark until a steady-state is reached, i.e., once the coefficient of variation (CoV) of 10 consecutive iterations falls below a predefined threshold, which is 0.02 in our case.
(2) For the JVM invocation i, compute the mean execution time of those 10 steady iterations, and denote it as $\bar{x}_i$.
(3) Repeat Step (1) and (2) for 10 times and compute the overall mean $\bar{x} = \frac{\sum_{i=1}^{10} \bar{x}_i}{10}$. Finally, we report $\bar{x}$ as the measurement result.

In the above measurement, Step (1) and (2) are designed to warm up the JVM, excluding factors like class loading and JIT compilation. These factors are less interesting to our evaluation because our main goal is to measure performance of variational execution. The coefficient of variation threshold is useful for controlling the effect of garbage collection. Step (3) is designed to minimize non-determinism of JIT compilation across JVM invocations, because JVM uses timer-based sampling to drive JIT optimization (e.g., which methods to optimize, at what level). Other main sources of non-determinism include thread scheduling and garbage collection. Thread scheduling is less of a concern for us because all programs except Jetty are single-threaded. Regarding Jetty, we configure Jetty to run a small server that has minimal thread scheduling. Georges et al. [2007]

Table 1. Statistics about benchmark programs and performance comparison among JVM, VarexJ and VarexC. Statistics include lines of code, number of (boolean) options, and number of valid configurations. Numbers in bold denote the cases where VarexC or VarexJ outperforms brute force execution. The last three columns denote the relative speedup or slowdown.

* Checkstyle contains a loop over a list of many optional elements. For VarexJ, we had to manually rewrite that loop to allow measurement (due to exponential behavior, we would run out of memory otherwise).

Subject	LOC	#Opt	#Config	μJVM (in ms)	max JVM (in ms)	VarexJ (in ms)	VarexC (in ms)	VarexJ/ VarexC	VarexJ/ maxJVM	VarexC/ maxJVM
Jetty	145, 421	7	128	949	1, 246	**166, 340**	**4, 660**	36x	133x	4x
Checkstyle	14, 950	141	$> 2^{135}$	811	946	***89, 366**	**3, 825**	23x	94x	4x
Prevayler	8, 975	8	256	13	44	**33, 124**	**725**	46x	753x	16x
QuEval	3, 109	23	940	0.03	0.38	2, 354	1, 244	2x	6, 195x	3, 274x
GPL	662	15	146	0.55	6.23	4, 691	479	10x	753x	479x
Elevator	730	6	20	0.03	0.07	45	7.88	6x	643x	113x
E-Mail	644	9	40	0.02	0.06	21	6.19	3x	350x	103x

recommends reporting a confidence interval instead of the mean alone. However, as we will show, the performance difference between our approach and VarexJ is so large that reporting confidence intervals is unnecessary. The difference is so obvious and the variation so small in comparison that statistical tests are not needed. Due to this large effect size, we omit confidence intervals for brevity.

Memory usage. To measure memory usage, we calculate the used heap space by calling APIs of `java.lang.Runtime` at *every method entry*, and then record the *maximum* heap space used throughout the entire JVM invocation. Even with this frequent sampling, we cannot guarantee accurate measurement of memory usage, largely because of the non-deterministic garbage collection and bulk memory allocation. Thus, the memory measurement is only useful for coarse-grained comparison. For VarexC and VarexJ, we perform each single measurement on a given subject program by executing it *once*. As a comparison goal, we also measure the memory usage of executing one representative configuration on a commodity JVM (reported as *JVM*). The representative configuration is chosen as a valid configuration with the most features enabled. Since VarexC and VarexJ explore the entire configuration space, their memory consumption is strictly larger than execution of a single configuration. To reduce noise, we repeat each measurement 10 times and report the average.

Sharing Efficiency. As discussed in Section 4.3, our approach is able to give static guarantees of optimal sharing to methods that satisfy certain conditions. To assess sharing for other methods, we monitor the sharing in our benchmark executions. Specifically, we collect traces of which VBlocks are executed under which conditions and subsequently analyze whether those traces were optimal, with regard to sharing. For each variational trace, we expand it into a set of all distinct concrete traces that it represents, and then compute the alignment of these concrete traces. Since an optimal alignment of n traces is NP-hard [Wang and Jiang 1994], we compute pairwise alignments between all distinct concrete executions using Needleman-Wunsch algorithm [Needleman and Wunsch 1970]. If the observed variational trace is longer than the longest pairwise alignment, we consider the sharing as not optimal. This pairwise approximation is conservative in that we may consider executions with optimal sharing as not optimal if the n-way alignment is longer than the longest pairwise alignment; conversely, if the variational trace is not longer than the longest pair-wise alignment we can be sure that the sharing is optimal. Our pairwise alignment approach sidesteps the need of computing optimal alignment, but it still has scalability issues if there are too many pairs, which happen sometimes in our evaluation. For those cases, we conservatively mark them as suboptimal.

6.2 Execution Time

Table 1 summarizes the performance results, showing that VarexC outperforms VarexJ by a factor between 2 to 46. Variational execution is obviously significantly slower than executing a single configuration (between 4 and 3200 times slower), but as configuration spaces grow exponentially, this slowdown is often practical to cover the entire space.

VarexC vs. VarexJ. Comparing VarexC and VarexJ, we can see that VarexC outperforms VarexJ in all cases, with a speedup of 2 to 46. To better understand the speedup, it is useful to divide our subject programs into two groups and discuss them separately.

QuEval, GPL, Elevator, E-mail are academic examples that only need basic language features, such as arithmetic computation and array operations. Thus, a comparison between VarexC and VarexJ on these programs reveals the performance gap between bytecode transformation and interpreter instrumentation. As we can see, we are up to 10 times faster than VarexJ, due to lower interpreter overhead and JVM optimizations. QuEval is dominated primarily by heavy computations with arrays with only moderate sharing that are expensive in both VarexC and VarexJ. In a micro-benchmark, we confirmed that sorting on an array of 1000 variational elements with VarexC is roughly 2 times faster than VarexJ, which likely explains the low performance difference for this program.

Jetty, Prevayler, Checkstyle are medium-sized real-world programs that are widely used in practice. These programs use various more advanced JVM features, including dynamic class loading (CheckStyle), network access (Jetty) and file access (Prevayler). Since VarexJ is built upon a research JVM, it inherits limitations from its underlying JVM in this regard, whereas code transformed with VarexC remains portable across JVMs.

VarexC vs. Individual Executions. To investigate how useful configuration-complete analyses are in practice, we compare VarexC (and for comparison also VarexJ) with the time it takes to execute individual configurations, both average configurations and worst-case configurations.

The overhead of variational execution is generally high, which is explained both by the instrumentation overhead (creating and propagating conditional values, boxing, control-flow indirections, SAT solving at runtime), and by doing the additional work of executing all configurations. The overhead is usually only justified for large configuration spaces, and so VarexC (as VarexJ) outperforms the brute-force execution of all configurations only for Jetty, CheckStyle, and Prevayler.

QuEval, GPL, Elevator, Email represent extreme cases where variations are used heavily. As we can see from Table 1, up to 940 configurations are encoded in merely 3, 109 lines of code for QuEval. When program variations (we called them features interchangeably) present compactly, the sharing of data and execution becomes less frequent, and thus explains why VarexC and VarexJ cannot outperform brute force because variational execution relies on sharing to be efficient. In fact, there is a loop in Checkstyle that causes state space explosion for VarexJ because of looping a list that has 2^{135} variants. VarexC uses a model class to handle this loop gracefully, as discussed in Section 5. However, unlike these extreme cases, programs in practice often adopt separation of concerns and thus features do not interact very heavily all the time [Meinicke et al. 2016].

Jetty, Prevayler, Checkstyle implement configuration options such that they are often orthogonal to each other or have relatively local effects, which facilitates sharing better. As we can see in Table 1, by exploiting sharing, the performance of VarexC for exploring the entire configuration space is even relatively close to executing only the slowest configuration, with a slowdown as small as a factor of 4.

Verdict. We argue that the runtime overhead of VarexC is reasonable except for one case (QuEval) where expensive array operations with little sharing dominate the performance. Runtime overhead does increase for the cases where interactions of variations are heavily used, but the overhead amortizes quickly in large configuration spaces, which grow exponentially with the number of

Table 2. Memory usage comparison of JVM, VarexJ and VarexC.

Subject	JVM (in MB)	VarexJ (in MB)	VarexC (in MB)	VarexJ/ VarexC	VarexJ/ JVM	VarexC/ JVM
Jetty	268	2, 739	648	4.2	10.2	2.4
Checkstyle	504	1, 106	835	1.3	2.2	1.7
Prevayler	65	1, 378	288	4.8	21.1	4.4
QuEval	59	301	282	1.1	5.1	4.8
GPL	141	342	151	2.3	2.4	1.1
Elevator	58	92	67	1.4	1.6	1.2
E-Mail	59	67	67	1.0	1.1	1.1

options, unless all options interact. More importantly, research shows that interactions do not increase with the worst-case exponential behavior in most cases [Meinicke et al. 2016; Reisner et al. 2010]. Even the academic programs that are designed to interact heavily are still well-behaved with plenty of sharing despite many interactions. Finally, we argue that the overhead is worthwhile if we consider the ability to identify all interactions among all options, for which the alternative is sampling only a small set of configurations.

In summary, VarexC outperforms VarexJ with a speedup of 2 to 46 times. The performance gain comes from various factors, including further optimizations at low level and portability to mature JVM implementations, all of which benefit from our strategy of transforming bytecode instead of modifying a language interpreter. Moreover, VarexC is performant and efficient for practical use in analyzing the whole configuration space of programs.

6.3 Memory Usage

Table 2 summarizes the memory usage results, showing that VarexC is more memory efficient than VarexJ in all cases except for a tie in E-Mail. Conceptually, VarexC and VarexJ perform a similar computation, so the extra memory consumed by VarexJ could result from two main aspects: less efficient sharing in data and the overhead of the underlying meta-circular interpreter. As the differences are fairly consistent across benchmarks, we attribute most efficiency gains to the interpreter's overhead rather than to differences in sharing. Both VarexC and VarexJ, as expected, consume more memory when compared to the execution of a single configuration, with the gaps noticeably smaller for VarexC. The memory overhead of VarexC largely comes from analyzing other configurations. We argue that the extra memory overhead shown in Table 2 is acceptable for modern machines.

In summary, VarexC is more memory efficient than VarexJ, due to more efficient sharing in data and less overhead from the implementation. Moreover, VarexC has the memory efficiency to analyze the entire configuration space in practice.

6.4 Sharing Efficiency

Table 3 shows how efficient our sharing of VBlocks is in practice. As we can see, we can make static guarantees for 89.7 percent of all the methods in our benchmark programs. When observing the executions, those methods with static guarantees account for 88.2 percent of the executed methods, and we observed that 99.8 percent for the remaining ones were optimal as well. The number of method executions that redundantly execute VBlocks with suboptimal sharing is minimal.

In summary, sharing in VarexC is efficient, with static guarantees to 89.7% *of all methods. For methods with no static guarantees, VarexC achieves runtime optimality for* 99.8% *of those method invocations.*

Table 3. Sharing efficiency of VarexC. We analyze methods both statically and at runtime. At runtime, we distinguish between method executions that are statically guaranteed to be optimal, that are dynamically observed to be optimal, and that are dynamically observed to be not optimal.

* Our alignment analysis has scalability issues with some variational traces of Checkstyle, mainly because there are too many features (up to 130) in each single trace, resulting into too expensive pairwise alignment. For those variational traces, we conservatively report them as non-optimal.

	Method analysis (static)		Method execution (dynamic)		
Subject	Guaranteed Optimal	No Guarantee	Guaranteed Optimal	Observed as Optimal	Observed as Non-Optimal
Jetty	2, 667	257	19, 043	3, 734	0
Checkstyle	2, 878	281	1, 992, 879	268, 689	*230
Prevayler	722	108	58, 274	5, 036	0
QuEval	458	103	57, 383	8, 920	267
GPL	244	44	34, 641	3, 476	0
Elevator	119	13	2, 453	218	1
E-Mail	314	40	2, 264	120	0
Total	7, 402	846	2, 166, 937	290, 193	498

7 RELATED WORK

We implement variational execution by transforming bytecode.

Variational execution. Variational execution is a technique to execute a program for different values while sharing common computations as far as possible. It has similarities with model checking and symbolic execution, but performs concrete executions, where multiple concrete values are distinguished with conditions external to the program, and focuses on maximizing sharing during the execution by storing variations in data locally and by aggressively merging control-flow differences. Variational execution has a number of existing and potential application scenarios in different lines of work. In each case, a program shall be executed for many similar inputs, typically to observe the similarities and differences among executions, often with the focus on interactions among multiple differences.

- A common use case is testing configurable systems, in which a single test case should be executed over a large configuration space. For example, Nguyen et al. [2014] used variational execution to render the content of WordPress while controlling how various plugins interact and affect the execution; Meinicke et al. [2016] and Kim et al. [2012] executed Java programs with configuration parameters (as used in our evaluation) to observe differences among different configurations. Given test cases to provide global or feature-specific specifications, variational execution can efficiently check such specifications by executing test cases over large configuration spaces [Kästner et al. 2012; Kim et al. 2012; Nguyen et al. 2014]. Soares et al. [2018] furthermore used differences among executions as clues to find suspicious feature interactions. Reisner et al. [2010] used symbolic execution to also detect feature interactions, which however required a lot of effort (80 machine weeks to symbolically execute 319 tests with less than 30 configuration options for 10KLOC programs) due to limited sharing abilities of symbolic execution [Meinicke et al. 2016].
- Austin and Flanagan [2012] uses variational execution (under the name faceted execution) to track information flows in a program. In this context, the program is evaluated with sensitive and nonsensitive values at the same time, where the equivalent of options are decisions who is allowed to see which value. In contrast to prior multi-execution work which observes differences between two executions, Austin's analysis based on variational execution can track interactions among multiple decisions. This line of work has been extended with models for variational database storage [Yang et al. 2016]. There are also libraries to enable developers to directly write

variational programs for this information-flow analysis, rather than relying on a variational execution engine [Austin et al. 2013; Schmitz et al. 2018, 2016].
- Variational execution can further be used to explain the differences in program executions among multiple inputs [Meinicke et al. 2018], in line with delta debugging [Kwon et al. 2016; Sumner and Zhang 2013; Zeller 2002].
- Variational execution is potentially useful for approaches that speculatively change source code or execution to evaluate the consequences. For example, mutation testing [Jia and Harman 2011] and generate-and-validate automatic program repair [Le Goues et al. 2012] typically try many small changes to the source code and re-execute the test suite for each change to evaluate test suite quality or find patches. Zhang et al. [2007] speculatively switches predicates in program and re-executes the program to detect execution omission errors. Brun et al. [2011] proactively merges different versions and repeatedly executes the test suite to detect collaboration conflicts early. By encoding changes as variations, variational execution can explore the effects of changes efficiently and uncover interesting interactions of changes [Wong et al. 2018].
- Finally, variational execution can be used to speed up similar computations if there is sufficient sharing to offset the overhead. For example, Sumner et al. [2011] shares similarities among executions of simulation workloads and computes with several values in parallel. Wang et al. [2017] shares executions of mutated programs with equivalence modulo states in the same process and forks new processes only if there are differences in program states after executing mutated statements. Tucek et al. [2009] executes patched and unpatched programs together to share redundant computations when testing a patch. Variational execution has the potential to scale such use cases to exploring interactions among multiple changes.

Variational execution is fundamentally different from traditional approaches of multi-execution [De Groef et al. 2012; Devriese and Piessens 2010; Hosek and Cadar 2013; Kolbitsch et al. 2012; Su et al. 2007] and delta debugging [Kwon et al. 2016; Sumner and Zhang 2013; Zeller 2002] that execute programs repeatedly (either variants of the program or the same program with different inputs) to compare those executions to identify, for example, information-flow issues or causes of bugs. These kinds of approaches execute programs repeatedly in parallel and align those executions either afterward or through probes at specific points of the executions. In contrast, variational execution exploits sharing and allows to observe differences among executions during the execution.

Ideas similar to variational execution can be found also in approaches for model checking and symbolic execution [d'Amorim et al. 2008; Sen et al. 2015; von Rhein et al. 2011], specifically concepts to store variations as local as possible to increase sharing and facilitate joining. Such tools can potentially be used for similar purposes when differences among inputs are modeled as symbolic decisions, but all other inputs are concrete. However, as Meinicke et al. [2016] has shown, current approaches are less effective at sharing than the aggressive sharing in variational execution.

Implementing Variational Execution. Existing variational execution approaches (and related approaches) are typically implemented by modifying the execution engine [Austin and Flanagan 2012; Barthe et al. 2012; Kästner et al. 2012; Kim et al. 2012; Maurer and Brumley 2012; Meinicke et al. 2016; Sen et al. 2015], typically research prototypes or metacircular interpreters that cause significant overhead and provide only limited support for all language features. Schmitz et al. [2018, 2016] provided a library for Haskell with which users can directly implement programs to use variational execution, similar to our example in Section 2.

Instead, we pursue an approach in which we transparently modify Java bytecode to achieve variational execution on a commodity JVM. Our approach was inspired by Phosphor [Bell and Kaiser 2014], a dynamic taint analysis for Java that tracks taints by instrumenting bytecode. In

contrast to Phosphor, our modifications are significantly more extensive, as we need not only track additional data, but entirely change how computations and control flow happen in the program. CROCHET allows to explore different inputs to the same function by modifying bytecode to perform checkpoints and rollbacks on the heap of a commodity JVM [Bell and Pina 2018]. Comparing to CROCHET, our approach can achieve a more fine-grained sharing of executions while exploring different alternative values. The only other approach to execute programs variationally with commodity infrastructure is the implementation behind Jeeves [Yang et al. 2016], that uses metaprogramming to achieve similar changes for a small subset of Python. Their transformations are incomplete and not described beyond their implementation for a small example program.

Quality assurance for configurable systems. A main goal of variational execution is testing configurable systems. There are a wide range of approaches to analyze configurable systems with large configuration spaces, typically focused on reusing test cases across product variants, on sampling and on static analysis [Engström and Runeson 2011; Medeiros et al. 2016; Nie and Leung 2011; Pohl et al. 2005; Thüm et al. 2014]. Sampling strategies analyze or execute a subset of configurations, but such analysis is neither exhaustive nor does it allow to easily compare executions [Nie and Leung 2011]. For static analyses (including type checking, model checking, and data-flow analysis), researchers have explored many sharing strategies to encode variability locally (e.g., alternative types for expressions), to reason about large configuration spaces with propositional formulas, and to join computations early [Liebig et al. 2013; Thüm et al. 2014]. In a sense, variational execution can be seen as a generalization of these sharing techniques for an interpreter [Kästner et al. 2012]. Bodden et al. [2013] and Dimovski et al. [2017] describe how to lift existing static analyses by providing a variational framework on how to execute them.

8 CONCLUSIONS

While variational execution has been applied in different areas, an efficient implementation is still missing for practical use. We propose to achieve variational execution by transforming programs at the bytecode level. Our approach is transparent to the developers, and has various advantages such as making use of underlying optimizations of the JVM and remaining portable to different JVMs. Our approach transforms individual instructions and modifies the control flow of methods to exploit sharing of common execution across configurations. Even with aggressive modification to the control flow decisions, we formally prove that our transformation to the control flow is correct for all cases, and optimal for a large subset of cases. We further optimize our implementation with two different optimizations, each of which optimizes our approach from different aspects. With an empirical evaluation on 7 highly configurable systems, we show that our approach is 2 to 46 times faster while saving up to 3 quarters of memory usage when compared to the state-of-the-art. A monitoring at runtime further confirms that we achieve 99.8% optimality for the methods that we cannot guarantee optimal sharing. Overall, our results indicate that our approach is useful for analyzing highly configurable systems in practice.

A ONLINE EXTENDED VERSION

The appendix is available online: https://arxiv.org/abs/1809.04193.

ACKNOWLEDGMENTS

This work has been supported in part by the NSF (awards 1318808, 1552944, and 1717022) and AFRL and DARPA (FA8750-16-2-0042). Lazarek was supported through Carnegie Mellon's Research Experiences for Undergraduates in Software Engineering. We thank Jonathan Bell for his advice on bytecode transformation.

REFERENCES

Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 165–178.

Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-Agnostic Programs. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, 15–26.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 50:1–50:39.

Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. 2012. Secure Multi-Execution through Static Program Transformation. In *Formal Techniques for Distributed Systems*. Springer, 186–202.

Jonathan Bell and Gail E. Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 83–101.

Jonathan Bell and Luís Pina. 2018. CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 17:1–17:31.

Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 355–364.

Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 168–178.

Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks* 41, 1 (2003), 115–141.

Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. 2008. Delta Execution for Efficient State-Space Exploration of Object-Oriented Programs. *IEEE Transactions on Software Engineering (TSE)* 34, 5 (2008), 597–613.

Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: A Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 748–759.

Dominique Devriese and Frank Piessens. 2010. Noninterference Through Secure Multi-Execution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 109–124.

Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, and Andrzej Brabrand, Clausand Wąsowski. 2017. Efficient Family-Based Model Checking via Variability Abstractions. *International Journal on Software Tools for Technology Transfer (STTT)* 19, 5 (2017), 585–603.

Emelie Engström and Per Runeson. 2011. Software Product Line Testing - A Systematic Mapping Study. *Information and Software Technology* 53, 1 (2011), 2–13.

Martin Erwig and Eric Walkingshaw. 2013. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering IV (GTTSE)*. Springer, 55–100.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA)*. ACM, 57–76.

Klaus Havelund and Thomas Pressburger. 2000. Model Checking JAVA Programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)* 2, 4 (2000), 366–381.

Petr Hosek and Cristian Cadar. 2013. Safe Software Updates via Multi-Version Execution. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 612–621.

Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering (TSE)* 37, 5 (2011), 649–678.

Christian Kästner. 2017. *Differential Testing for Variational Analyses: Experience from Developing KConfigReader*. Technical Report 1706.09357. arXiv.

Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. 2012. Toward Variability-Aware Testing. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 1–8.

Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. 2012. Shared Execution for Efficiently Testing Product Lines. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 221–230.

Dohyeong Kim, Yonghwi Kwon, William N. Sumner, Xiangyu Zhang, and Dongyan Xu. 2015. Dual Execution for On the Fly Fine Grained Execution Comparison. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 325–338.

Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-Cloaking Internet Malware. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 443–457.

Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2016. LDX: Causality Inference by Lightweight Dual Execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 503–515.

Lukas Lazarek. 2017. How to Efficiently Process 2^{100} List Variations. In *Proceedings Companion of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)*. ACM, 36–38.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering (TSE)* 38, 1 (2012), 54–72.

Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 81–91.

Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking Load-Time Configuration Options. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 445–456.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. *The Java® Virtual Machine Specification* (1 ed.). Addison-Wesley Professional.

Matthew Maurer and David Brumley. 2012. Tachyon: Tandem Execution for Efficient Live Patch Testing. In *Proceedings of the USENIX Security Symposium*. USENIX, 617–630.

Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 664–675.

Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 495–518.

Jens Meinicke, Chu-Pan Wong, Christian Kästner, and Gunter Saake. 2018. *Understanding Differences among Executions with Variational Traces*. Technical Report 1807.03837. arXiv.

Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 483–494.

Meng Meng, Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner. 2017. A Choice of Variational Stacks: Exploring Variational Data Structures. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS)*. ACM, 28–35.

Saul B. Needleman and Christian D. Wunsch. 1970. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology* 48, 3 (1970), 443–453.

Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 907–918.

A Nhlabatsi, R Laney, and B Nuseibeh. 2008. Feature Interaction: the Security Threat from within Software Systems. *Progress in Informatics* (2008), 75–89.

Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Computing Surveys (CUSR)* 43, 2 (2011), 11:1–11:29.

Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. 2010. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 445–454.

Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. 2018. Faceted Secure Multi Execution. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM.

Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. 2016. Faceted Dynamic Information Flow via Control and Data Monads. In *Proceedings of the International Conference on Principles of Security and Trust - Volume 9635*. Springer, 3–23.

Julia Schroeter, Malte Lochau, and Tim Winkelmann. 2012. Multi-Perspectives on Feature Models. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 252–268.

Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 842–853.

Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. 2018. VarXplorer: Lightweight Process for Dynamic Analysis of Feature Interactions. In *Proceedings of the International Workshop on*

Variability Modelling of Software-Intensive Systems (VAMOS). ACM, 59–66.

Ya-Yunn Su, Mona Attariyan, and Jason Flinn. 2007. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. ACM, 237–250.

William N. Sumner, Tao Bao, Xiangyu Zhang, and Sunil Prabhakar. 2011. Coalescing Executions for Fast Uncertainty Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 581–590.

William N. Sumner and Xiangyu Zhang. 2013. Comparative Causality: Explaining the Differences between Executions. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 272–281.

Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 6:1–6:45.

Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. 2009. Efficient Online Validation with Delta Execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 193–204.

Alexander von Rhein, Sven Apel, and Franco Raimondi. 2011. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In *Proceedings of the Java Pathfinder Workshop*.

Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*. ACM, 213–226.

Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster Mutation Analysis via Equivalence Modulo States. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 295–306.

Lusheng Wang and Tao Jiang. 1994. On the Complexity of Multiple Sequence Alignment. *Journal of Computational Biology* 1 (1994), 337–348.

Chu-Pan Wong, Jens Meinicke, and Christian Kästner. 2018. Beyond Testing Configurable Systems: Applying Variational Execution to Automatic Program Repair and Higher Order Mutation Testing. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering - New Ideas and Emerging Results Track (ESEC/FSE-NIER)*. ACM.

Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-backed Applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 631–647.

Andreas Zeller. 2002. Isolating Cause-Effect Chains From Computer Programs. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*. ACM, 1–10.

Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. 2007. Towards Locating Execution Omission Errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 415–424.